Int	::=	the domain of (unbounded) integer numbers, with usual operations on them
Bool	::=	the domain of booleans
VarId	::=	standard identifiers
AExp	::=	*Int* | *VarId* | *AExp* `+` *AExp* | *AExp* `/` *AExp*
BExp	::=	*Bool* | *AExp* `<=` *AExp* | `not` *BExp* | *BExp* `and` *BExp*
Stmt	::=	`skip` | *VarId* `:=` *AExp* | *Stmt* `;` *Stmt* |
		`if` *BExp* `then` *Stmt* `else` *Stmt* | `while` *BExp* `do` *Stmt*
Pgm	::=	`vars` **List**`{`*VarId*`}` `;` *Stmt*

Figure 3.1: Syntax of IMP, a small imperative language, using algebraic BNF

3.1 IMP: A Simple Imperative Language

To illustrate the various operational semantics styles, we have chosen a small imperative language, called IMP. The IMP language has arithmetic expressions which include the domain of arbitrarily large integer numbers, boolean expressions, assignment statements, conditional statements, while loop statements, and sequential composition of statements. All variables used in an IMP program are expected be declared at the beginning of the program, can only hold integer values (for simplicity, there are no boolean type variables in IMP), and are instantiated with default value 0.

3.1.1 IMP Syntax

We here define the syntax of IMP, first using the Backus-Naur form (BNF) notation for context-free grammars and then using the alternative and completely equivalent mixfix algebraic notation (see Section 2.5). The latter is in general more appropriate for semantic developments of a language.

IMP Syntax as a Context-Free Grammar

The syntax of IMP using the algebraic BNF notation is depicted in Figure 3.1. The only "algebraic" feature is the use of **List**`{`*VarId*`}` for variable declarations (last production), which in this case is clear: one can declare a comma-separated list of variables. To stay more conventional in notation, we refrained from replacing the productions "*Stmt* ::= `skip` | *Stmt* `;` *Stmt*" by the algebraic production "*Stmt* ::= $\mathbf{List}^{\texttt{skip}}_{\texttt{;}}$`{`*Stmt*`}`" which captures the idea of statement sequentialization more naturally; indeed, the former yields ambiguous parsing (luckily, the semantics of statement sequential composition will be such that the parsing ambiguity is irrelevant, but that is not always the case).

The IMP language constructs have their usual imperative meaning. For diversity and demonstration purposes, when giving the various semantics of IMP we will assume that `+` is *non-deterministic* (it evaluates the two subexpressions in any order, possibly interleaving their corresponding evaluation steps), `/` is non-deterministic and *partial* (it will stuck the program when a division by zero takes place), `<=` is *left-right sequential* (it first evaluates the left subexpression and then the right subexpression), and that `and` is left-right sequential and *short-circuited* (it first evaluates the left subexpression and then it conditionally evaluates the right only if the left evaluated to true).

To expose some of the limitations of the existing operational approaches, in Section 3.8 we extend IMP with expression side effects (an increment operation on variables), with abrupt termination (a

sorts:
$Int,\ Bool,\ Var,\ AExp,\ BExp,\ Stmt,\ Pgm$
subsorts:
$Int,\ Var < AExp$
$Bool < BExp$
operations:

$$\begin{array}{rcl}
\texttt{+} & : & AExp \times AExp \to AExp \\
\texttt{/} & : & AExp \times AExp \to AExp \\
\texttt{<=} & : & AExp \times AExp \to BExp \\
\texttt{and} & : & BExp \times BExp \to BExp \\
\texttt{not}_ & : & BExp \to BExp \\
\texttt{skip} & : & \to Stmt \\
\texttt{:=} & : & Var \times AExp \to Stmt \\
\texttt{;} & : & Stmt \times Stmt \to Stmt \\
\texttt{if}_\texttt{then}_\texttt{else}_ & : & BExp \times Stmt \times Stmt \to Stmt \\
\texttt{while}_\texttt{do}_ & : & BExp \times Stmt \to Stmt \\
\texttt{vars}_\texttt{;}_ & : & \mathbf{List}\{VarId\} \times Stmt \to Pgm
\end{array}$$

Figure 3.2: Syntax of IMP as an algebraic signature

halt statement), and with dynamic threads. The extension of IMP with side effects, in particular, makes the various evaluation strategies of `+`, `<=` and `and` semantically relevant.

We will assume available the domains of integers and booleans, as well as basic operations on them which are clearly tagged (e.g., "$+_{Int}$" for addition of integer numbers) to distinguish them from homonymous operations which are IMP language constructs.

We may tacitly use the following naming conventions for terms or variables throughout the remainder of this chapter: $x, X \in Var$; $a, A \in AExp$; $b, B \in BExp$; $s, S \in Stmt$; $i, I \in Int$; $t, T \in Bool$; $p, P \in Pgm$. Any of these can be primed or indexed.

IMP Syntax as an Algebraic Signature

Following the relationship between the CFG and the mixfix algebraic notations explained in Section 2.5, the BNF syntax in Figure 3.1 can be associated the entirely equivalent algebraic signature in Figure 3.2 with one (mixfix) operation per production: the terminals mixed with underscores form the name of the operation and the non-terminals give its arity. This signature is easy to define in any rewrite engine or theorem prover; moreover, it can also be defined as a data-type or corresponding structure in any programming language. We next show how it can be defined in Maude.

☆ Definition of IMP Syntax in Maude

Using the Maude notation for algebraic signatures, the algebraic signature in Figure 3.2 can yield the Maude syntax module in Figure 3.3. We have additionally picked some appropriate precedences and formatting attributes for the various language syntactic constructs.

The module `IMP-SYNTAX` in Figure 3.3 imports three "builtin" modules, namely: `INT`, which we assume it provides a sort `Int`; `BOOL`, which we assume provides a sort `Bool`; and `VAR` which

```
mod IMP-SYNTAX is including INT + BOOL + VAR .
--- AExp
  sort AExp .  subsorts Int VarId < AExp .
  op _+_ : AExp AExp -> AExp [prec 33 gather (E e) format (d b o d)] .
  op _/_ : AExp AExp -> AExp [prec 31 gather (E e) format (d b o d)] .
--- BExp
  sort BExp .  subsort Bool < BExp .
  op _<=_ : AExp AExp -> BExp  [prec 37 format (d b o d)] .
  op not_ : BExp -> BExp [prec 53 format (b o d)] .
  op _and_ : BExp BExp -> BExp [prec 55 format (d b o d)] .
--- Stmt
  sort Stmt .
  op skip : -> Stmt [format (b o)] .
  op _:=_ : VarId AExp -> Stmt [prec 40 format (d b o d)] .
  op _;_ : Stmt Stmt -> Stmt [prec 60 gather (e E) format (d b noi d)] .
  op if_then_else_ : BExp Stmt Stmt -> Stmt [prec 59 format (b o bni n++i bn--i n++i --)] .
  op while_do_ : BExp Stmt -> Stmt [prec 59 format (b o d n++i --)] .
--- Pgm
  sort Pgm .
  op vars_;_ : List{VarId} Stmt -> Pgm [prec 70 format (nb o d ni d)] .
endm
```

Figure 3.3: IMP syntax as an algebraic signature in Maude. This definition assumes appropriate modules `INT`, `BOOL` and `VAR` defining corresponding sorts `Int`, `Bool`, and `VarId`, respectively.

we assume provides a sort `VarId`. We do not give the precise definitions of these modules here, particularly because one may have many different ways to do it. In our examples from here on in the rest of the chapter we assume that `INT` contains all the integer numbers as constants of sort `Int`, that `BOOL` contains the constants `true` and `false` of sort `Bool`, and that `VAR` contains all the letters in the alphabet as constants of sort `Var`. Also, we assume that the module `INT` comes equipped with as many "builtin" operations on integers as needed.

To avoid operator name conflicts caused by Maude's operator overloading capabilities, we urge the reader *not* to use the Maude builtin `INT` and `BOOL` modules, but instead to overwrite them. Appendix A.1 shows one possible way to do this in Maude 2.4: we define new modules `INT` and `BOOL` "hooked" to the builtin integer and boolean values but defining only a subset of operations on them and with names clearly tagged as discussed above, e.g., `_+Int_ : Int Int -> Int`, etc.

Recall from Sections 2.4 and 2.8 that lists, sets, bags and maps are trivial algebraic structures which can be easily defined in Maude; consequently, we take the freedom to use them without definition whenever needed, as we did with using the sort `List{VarId}` in Figure 3.3.

To test the syntax, one can now parse various IMP programs, such as:

```
Maude> parse
     vars n, s ;
     n := 100 ;
     s := 0 ;
     while not(n <= 0) do (
       s := s + n ;
       n := n + -1
     )
.
```

Now it is a good time to define a module, say `IMP-PROGRAMS`, containing as many IMP programs

```
mod IMP-PROGRAMS is including IMP-SYNTAX .
ops sumPgm collatzPgm countPrimesPgm : -> Pgm .
ops collatzStmt multiplicationStmt primalityStmt : -> Stmt .
eq sumPgm = (
     vars n, s ;
     n := 100 ;  s := 0 ;
     while not(n <= 0) do (
       s := s + n ; n := n + -1
     )     ) .
eq collatzStmt = (
     while not (n <= 1) do (
       s := s + 1 ; q := n / 2 ; r := q + q + 1 ;
       if r <= n
         then n := n + n + n + 1
         else n := q
     )     ) .
eq collatzPgm = (
   vars m, n, q, r, s ;
   m := 10 ; s := 0 ;
   while not (m <= 2) do (
     n := m ; m := m + -1 ;
     collatzStmt
   )     ) .
eq multiplicationStmt = ( --- fast multiplication (base 2) algorithm
  z := 0 ;
  while not(x <= 0) do (
    q := x / 2 ; r := q + q + 1 ;
    if r <= x then z := z + y                    --- if x % 2 == 1
    else skip ;
    x := q ;
    y := y + y
  )     ) .
eq primalityStmt = (
  i := 2 ; q := n / i ; t := 1 ;
  while (i <= q and 1 <= t) do (
    x := i ; y := q ;
    multiplicationStmt ;
    if n <= z then t := 0 else (
      i := i + 1 ; q := n / i
    )
  )     ) .
eq countPrimesPgm = (
   vars i, m, n, q, r, s, t, x, y, z ;
   m := 10 ; s := 0 ; n := 2 ;
   while n <= m do (
     primalityStmt ;
     if 1  <= t then s := s + 1 else skip ;
     n := n + 1
   )     ) .
endm
```

Figure 3.4: IMP programs defined in a Maude module `IMP-PROGRAMS`

as one bears to write. Figure 3.4 shows such a module containing several IMP programs. Note that we took advantage of Maude's rewriting capabilities to save space and reuse some of the defined fragments of programs as "macros". Defining such a module helps us to test the desired language syntax (Maude will report errors if the programs that appear in the right-hand-sides of the equations are not parsable), and will also help us later on to test the various semantics that we will define.

3.1.2 IMP State

Any operational semantics of IMP needs some appropriate notion of *state*, which is expected to map program variables to integer values. Moreover, since IMP disallows uses of undeclared variables, it suffices for the state of a given program to only map the declared variables to values and stay undefined and disallow updates of variables which were not declared.

Fortunately, all these desired IMP state operations correspond to conventional mathematical operations on *partial finite-domain functions* from variables to integers in $[\mathit{VarId} \rightharpoonup \mathit{Int}]^{\mathit{finite}}$ (see Section 2.1.2) or, equivalently, to equationally defined structures of sort $\mathbf{Map}\{\mathit{VarId} \mapsto \mathit{Int}\}$ (see Section 2.3.2 for details on the notation and the equivalence); we let *State* be an alias for the map sort above. From a rewriting logic semantic point of view, the equations defining such map structure are invisible: semantic transitions that are part of various IMP semantics will be performed *modulo* these equations. In other words, state lookup and update operations will not count as computational steps, so they will not interfere with or undesirably modify the intended computational granularity of the defined language (IMP in this case).

We let σ, σ', σ_1, etc., range over states. By defining IMP states as partial finite-domain functions $\sigma : \mathit{Var} \rightharpoonup \mathit{Int}$, we have a very natural notion of undefinedness for a variable that has not been declared and thus initialized in a state: state σ is considered *undefined* in a variable x if and only if $x \notin \mathit{Dom}(\sigma)$. We may use the terminology *state lookup* for the operation $\_(\_) : \mathit{State} \times \mathit{VarId} \to \mathit{Int}$, the terminology *state update* for the operation $\_[\_/\_] : \mathit{State} \times \mathit{Int} \times \mathit{VarId} \to \mathit{State}$, and the terminology *state initialization* for the operation $\_ \mapsto \_ : \mathbf{List}\{\mathit{VarId}\} \to \mathit{Int}$; recall from Sections 2.1.2 and 2.3.2 that the latter operation builds a partial function mapping each element in the first list argument (these elements are expected to be distinct) to the element given as second argument.

☆ Definition of IMP State in Maude

Figure 3.5 adapts the generic Maude definition of partial finite-domain functions in Figure 2.2 for our purpose here: the generic sorts `A` (for the source) and `B` (for the target) are replaced by `VarId` and `Int`, respectively, and the definition of state update is simplified to only update variables that are already in the domain of the map (this suffices for IMP). The only reason for which we bother to give this obvious module is because we want the various subsequent semantics of the IMP language, all of them including the module `STATE` in Figure 3.5, to be self-contained and executable in Maude by simply executing all the Maude code in the figures in this chapter.

3.2 Natural, or Big-Step Operational Semantics

Known also under the names *natural semantics*, *relational semantics* and *evaluation semantics*, big-step semantics is "the most denotational" of the operational semantics. One can view big-step definitions as definitions of functions, or more generally of relations, interpreting each language construct in an appropriate domain. Big-step semantics is so natural, that one is encouraged to

```
mod STATE is including INT + VAR .
  sort State .
  op _|->_ : List{VarId} Int -> State [prec 0] .
  op .State : -> State .
  op _,_ : State State -> State [assoc comm id: .State format(d s s d)] .
  op _(_) : State VarId -> [Int] [prec 0] .        --- lookup
  op _[_/_] : State Int VarId -> State [prec 0] .  --- update

  var Sigma : State .  var X X' : VarId .  var Xl : List{VarId} .  var I I' : Int .

  eq (Sigma, X |-> I)(X) = I .
  eq (Sigma, X |-> I)[I' / X ] = (Sigma, X |-> I') .
  eq (X,X',Xl) |-> I = X |-> I, X' |-> I, Xl |-> I .
  eq .List{VarId} |-> I = .State .
endm
```

Figure 3.5: The IMP state defined in Maude

use it whenever possible. Unfortunately, as discussed in Section 3.9, big-step semantics has a series of drawbacks making it inconvenient or impossible to use in many situations, for example when defining languages with control-intensive features or concurrency.

Under big-step semantics, the sequents are relations of configurations, typically written $C \Rightarrow R$ or $C \Downarrow R$, with the meaning that R is the configuration obtained after the (complete) evaluation of C. In this book we prefer the notation $C \Downarrow R$. A *big-step rule* therefore has the form

$$\frac{C_1 \Downarrow R_1,\ C_2 \Downarrow R_2,\ \ldots,\ C_n \Downarrow R_n}{C \Downarrow R}$$

where C, C_1, C_2, ..., C_n are configurations holding fragments of program together with all the needed semantic components, and R, R_1, R_2, ..., R_n are *result configurations*, i.e., configurations which cannot be advanced anymore. A big-step semantics describes in a divide-and-conquer manner how final evaluation results of language constructs can be obtained by combining the evaluation results of their syntactic counterparts (subexpressions, substatements, etc.). For example, the big-step semantics of addition in IMP is

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle,\quad \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 + a_2, \sigma\rangle \Downarrow \langle i_1 +_{Int} i_2\rangle}$$

Here, the meaning of a relation $\langle a, \sigma\rangle \Downarrow \langle i\rangle$ is that arithmetic expression a is evaluated in state σ to integer i. If expression evaluation has side-effects, then one has to also include a state in the right configurations, so they become of the form $\langle i, \sigma\rangle$ instead of $\langle i\rangle$, as discussed in Section 3.9. It is common in big-step semantics to not wrap single values in configurations, that is, to write $\langle a, \sigma\rangle \Downarrow i$ instead of $\langle a, \sigma\rangle \Downarrow \langle i\rangle$ and similarly for all the other sequents. In fact, the original notation for big-step sequents, which was given in the context of a pure language (with no side effects) with only a state needed as semantic infrastructure, the notation for big-step sequents was $\sigma \vdash a \Rightarrow i$, with the meaning that "in state (or environment) σ, a evaluates to i". Variants of big-step decorating the symbols $\vdash$ or $\Rightarrow$ with various semantic data or labels can also be found in the literature.

For the sake of a uniform notation, in particular when transiting from languages whose expressions have no side effects to languages whose expressions do have side effects (as we do in Section 3.9), we

sorts:

Configuration

operations:

$$\begin{array}{rcl} \langle \_, \_ \rangle & : & \mathit{AExp} \times \mathit{State} \to \mathit{Configuration} \\ \langle \_ \rangle & : & \mathit{Int} \to \mathit{Configuration} \\ \langle \_, \_ \rangle & : & \mathit{BExp} \times \mathit{State} \to \mathit{Configuration} \\ \langle \_ \rangle & : & \mathit{Bool} \to \mathit{Configuration} \\ \langle \_, \_ \rangle & : & \mathit{Stmt} \times \mathit{State} \to \mathit{Configuration} \\ \langle \_ \rangle & : & \mathit{State} \to \mathit{Configuration} \\ \langle \_ \rangle & : & \mathit{Pgm} \to \mathit{Configuration} \end{array}$$

Figure 3.6: IMP big-step configurations as an algebraic signature.

prefer to always write big-step sequents as $C \Downarrow R$, and always use the angle brackets to surround both configurations involved. This solution is the most general; for example, any additional semantic data or labels that one may need in a big-step definition can be uniformly included as additional components in the configurations.

3.2.1 IMP Configurations for Big-Step Operational Semantics

For the big-step semantics of the simple language IMP, we only need very simple configurations. We follow the comma-and-angle-bracket notational convention, that is, we separate the configuration components by commas and then enclose the entire list with angle brackets. For example, $\langle a, \sigma \rangle$ is a configuration containing an arithmetic expression a and a state σ, and $\langle b, \sigma \rangle$ is a configuration containing a boolean expression b and a state σ. Some configurations may not need a state while others may not need the code. For example, $\langle i \rangle$ is a configuration holding only the integer number i that can be obtained as a result of evaluating an arithmetic expression, while $\langle \sigma \rangle$ is a configuration holding only one state σ that can be obtained after evaluating a statement. Configurations can therefore be of different types and need not necessarily have the same number of components. Here are all the configuration types needed for the big-step semantics of IMP:

- $\langle a, \sigma \rangle$ grouping arithmetic expressions a and states σ;
- $\langle i \rangle$ holding integers i;
- $\langle b, \sigma \rangle$ grouping boolean expressions b and states σ;
- $\langle t \rangle$ holding truth values $t \in \{\mathit{true}, \mathit{false}\}$;
- $\langle s, \sigma \rangle$ grouping statements s and states σ;
- $\langle \sigma \rangle$ holding states σ;
- $\langle p \rangle$ holding programs p;

IMP Big-Step Configurations as an Algebraic Signature

The configurations above were defined rather informally as tuples of syntax and/or states. There are many ways to rigorously formalize them, all building upon some formal definition of state (besides

```
mod IMP-CONFIGURATIONS-BIGSTEP is including IMP-SYNTAX + STATE .
  sort Configuration .
  op <_,_> : AExp State -> Configuration .
  op   <_> : Int -> Configuration .
  op <_,_> : BExp State -> Configuration .
  op   <_> : Bool -> Configuration .
  op <_,_> : Stmt State -> Configuration .
  op   <_> : State -> Configuration .
  op   <_> : Pgm -> Configuration .
endm
```

Figure 3.7: The IMP configurations for big-step operational semantics defined in Maude

IMP syntax). Since we have already defined states as partial finite-domain functions (Section 3.1.2) and have already shown how partial finite-domain functions can be formalized as algebraic specifications (Section 2.3.2), we also formalize configurations algebraically.

Figure 3.6 shows an algebraic signature defining the IMP configurations needed for the subsequent big-step operational semantics. For simplicity, we preferred to explicitly define each type of needed configuration. Consequently, our configurations definition in Figure 3.6 may be more verbose than an alternative polymorphic definition, but we believe that it is clearer for this simple language. We assumed that the sorts *AExp*, *BExp*, *Stmt*, *Pgm* and *State* come from algebraic definitions of the IMP syntax and state, like those in Sections 3.1.1 and 3.1.2; recall that the latter adapted the algebraic definition of partial functions in Section 2.3.2 (see Figure 2.1) as explained in Section 3.1.2.

☆ Maude Definition of IMP Configurations for Big-Step Operational Semantics

Figure 3.7 needs no explanation. It is a straightforward translation into Maude of the algebraic signature in Figure 3.6.

3.2.2 The Big-Step Operational Semantics Rules of IMP

Figure 3.8 shows all the rules in our IMP big-step operational semantics proof system. Recall that the role of a proof system is to prove, or derive, facts. The facts that our proof system will derive have the forms $\langle a, \sigma\rangle \Downarrow \langle i\rangle$, $\langle b, \sigma\rangle \Downarrow \langle t\rangle$, $\langle s, \sigma\rangle \Downarrow \langle \sigma'\rangle$, and $\langle p\rangle \Downarrow \langle \sigma\rangle$ where a ranges over *AExp*, b over *BExp*, s over *Stmt*, p over *Pgm*, i over *Int*, t over *Bool*, and σ and σ' over *State*.

Informally[1], the meaning of derived triples of the form $\langle a, \sigma\rangle \Downarrow \langle i\rangle$ is that the arithmetic expression a evaluates/executes/transits to the integer i in state σ; the meaning of $\langle b, \sigma\rangle \Downarrow \langle t\rangle$ is similar but with boolean values instead of integers. The reason for which it suffices to derive such simple facts is because the evaluation of expressions in our simple IMP language is side-effect-free. When we add the increment operation "`++ x`" in Section 3.9, we will have to change the big-step semantics to work with 4-tuples of the form $\langle a, \sigma\rangle \Downarrow \langle i, \sigma'\rangle$ and $\langle b, \sigma\rangle \Downarrow \langle t, \sigma'\rangle$ instead. The meaning of $\langle s, \sigma\rangle \Downarrow \langle \sigma'\rangle$ is that the statement s takes state σ to state σ'. Finally, the meaning of pairs $\langle p\rangle \Downarrow \langle \sigma\rangle$ is that the program p yields state σ when executed in the initial state.

In the case of our simple IMP language, the transition relation is going to be *deterministic*, in the sense that $i_1 = i_2$ whenever $\langle a, \sigma\rangle \Downarrow \langle i_1\rangle$ and $\langle a, \sigma\rangle \Downarrow \langle i_2\rangle$ can be deduced (and similarly for boolean

[1]Formal definitions of these concepts can only be given after one has a formal language definition. We formally define the notions of evaluation and termination in the context of the IMP language in Definition 16.

$$\langle i, \sigma\rangle \Downarrow \langle i\rangle \qquad \text{(BigStep-Int)}$$

$$\langle x, \sigma\rangle \Downarrow \langle \sigma(x)\rangle \qquad \text{(BigStep-Lookup)}$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle,\ \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \texttt{ + } a_2, \sigma\rangle \Downarrow \langle i_1\ +_{Int}\ i_2\rangle} \qquad \text{(BigStep-Add)}$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle,\ \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \texttt{ / } a_2, \sigma\rangle \Downarrow \langle i_1\ /_{Int}\ i_2\rangle} \text{, where } i_2 \neq 0 \qquad \text{(BigStep-Div)}$$

$$\langle t, \sigma\rangle \Downarrow \langle t\rangle \qquad \text{(BigStep-Bool)}$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle,\ \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \texttt{ <= } a_2, \sigma\rangle \Downarrow \langle i_1\ \leq_{Int}\ i_2\rangle} \qquad \text{(BigStep-Leq)}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \texttt{true}\rangle}{\langle \texttt{not } b, \sigma\rangle \Downarrow \langle \texttt{false}\rangle} \qquad \text{(BigStep-Not-True)}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \texttt{false}\rangle}{\langle \texttt{not } b, \sigma\rangle \Downarrow \langle \texttt{true}\rangle} \qquad \text{(BigStep-Not-False)}$$

$$\frac{\langle b_1, \sigma\rangle \Downarrow \langle \texttt{false}\rangle}{\langle b_1 \texttt{ and } b_2, \sigma\rangle \Downarrow \langle \texttt{false}\rangle} \qquad \text{(BigStep-And-False)}$$

$$\frac{\langle b_1, \sigma\rangle \Downarrow \langle \texttt{true}\rangle,\ \langle b_2, \sigma\rangle \Downarrow \langle t\rangle}{\langle b_1 \texttt{ and } b_2, \sigma\rangle \Downarrow \langle t\rangle} \qquad \text{(BigStep-And-True)}$$

$$\langle \texttt{skip}, \sigma\rangle \Downarrow \langle \sigma\rangle \qquad \text{(BigStep-Skip)}$$

$$\frac{\langle a, \sigma\rangle \Downarrow \langle i\rangle}{\langle x \texttt{ := } a, \sigma\rangle \Downarrow \langle \sigma[i/x]\rangle} \qquad \text{(BigStep-Asgn)}$$

$$\frac{\langle s_1, \sigma\rangle \Downarrow \langle \sigma_1\rangle,\ \langle s_2, \sigma_1\rangle \Downarrow \langle \sigma_2\rangle}{\langle s_1 \texttt{;} s_2, \sigma\rangle \Downarrow \langle \sigma_2\rangle} \qquad \text{(BigStep-Seq)}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \texttt{true}\rangle,\ \langle s_1, \sigma\rangle \Downarrow \langle \sigma_1\rangle}{\langle \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2, \sigma\rangle \Downarrow \langle \sigma_1\rangle} \qquad \text{(BigStep-If-True)}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \texttt{false}\rangle,\ \langle s_2, \sigma\rangle \Downarrow \langle \sigma_2\rangle}{\langle \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2, \sigma\rangle \Downarrow \langle \sigma_2\rangle} \qquad \text{(BigStep-If-False)}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \texttt{false}\rangle}{\langle \texttt{while } b \texttt{ do } s, \sigma\rangle \Downarrow \langle \sigma\rangle} \qquad \text{(BigStep-While-False)}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \texttt{true}\rangle,\ \langle s \texttt{;while } b \texttt{ do } s, \sigma\rangle \Downarrow \langle \sigma'\rangle}{\langle \texttt{while } b \texttt{ do } s, \sigma\rangle \Downarrow \langle \sigma'\rangle} \qquad \text{(BigStep-While-True)}$$

$$\frac{\langle s, xl \mapsto 0\rangle \Downarrow \langle \sigma\rangle}{\langle \texttt{ vars } xl \texttt{ ; } s\rangle \Downarrow \langle \sigma\rangle} \qquad \text{(BigStep-Vars)}$$

Figure 3.8: BigStep(IMP) — IMP Big-Step Operational Semantics ($i, i_1, i_2 \in \mathit{Int}$; $x \in \mathit{VarId}$; $\sigma, \sigma', \sigma_1, \sigma_2 \in \mathit{State}$; $a, a_1, a_2 \in \mathit{AExp}$; $b, b_1, b_2 \in \mathit{BExp}$; $t \in \mathit{Bool}$; $s, s_1, s_2 \in \mathit{Stmt}$; $xl \in \mathbf{List}\{\mathit{VarId}\}$).

expressions, statements and programs). However, in the context of nondeterministic languages, triples $\langle a, \sigma\rangle \Downarrow \langle i\rangle$ state that *a may possibly* evaluate to i in state σ, but it may also evaluate to other integers (and similarly for boolean expressions, statements and programs).

The proof system in Figure 3.8 contains one or two rules for each language construct, capturing its intended transition relation. Recall from Section 2.2 that proof rules are in fact *rule schemas*, that is, they correspond to (recursively enumerable) sets of *rule instances*, one for each concrete instance of the rule *parameters* (i.e., a, b, σ, etc.). We next discuss each of the rules in Figure 3.8.

The rules (BigStep-Lookup) and (BigStep-Int) define the obvious semantics of variable lookup and integers; these rules are unconditional because variables and integers are atomic expressions, so one does not need to evaluate any other sub-expression in order to evaluate them.

The rule (BigStep-Add) has already been discussed at the beginning of Section 3.2. (BigStep-Div) is similar, but note that it has a *side condition*. As discussed in Section 2.2, the role of side conditions is to filter out undesirable instances of the rule. Note that we chose not to "short-circuit" the division operation when a_1 evaluates to 0. Consequently, no matter whether a_1 evaluates to 0 or not, a_2 is still expected to produce a correct value in order for the rule BigStep-Div to be applicable (e.g., a_2 cannot perform a division by 0).

Exercise 32. *Change the rule* BigStep-Div *so that division short-circuits when* a_1 *evaluates to* 0. *(Hint: may need to replace it with two rules, like for the semantics of conjunction).*

Before we continue with the remaining rules, let us clarify, using concrete examples, what it means for rule schemas to admit multiple instances and how these can be used to derive proofs. For example, a possible instance of rule (BigStep-Add) can be the following (assume that $x, y \in \mathit{VarId}$):

$$\frac{\langle 1, (x \mapsto 0, y \mapsto 0)\rangle \Downarrow \langle 1\rangle,\ \langle 2, (x \mapsto 0, y \mapsto 0)\rangle \Downarrow \langle 2\rangle}{\langle 1 + 2, (x \mapsto 0, y \mapsto 0)\rangle \Downarrow \langle 3\rangle}$$

Another instance of rule (BigStep-Add) is the following, which, of course, seems problematic:

$$\frac{\langle 1, (x \mapsto 0, y \mapsto 0)\rangle \Downarrow \langle 1\rangle,\ \langle 2, (x \mapsto 0, y \mapsto 0)\rangle \Downarrow \langle 9\rangle}{\langle 1 + 2, (x \mapsto 0, y \mapsto 0)\rangle \Downarrow \langle 10\rangle}$$

The rule above is indeed a correct instance of (BigStep-Add), but, however, one will never be able to infer $\langle 2, (x \mapsto 0, y \mapsto 0)\rangle \Downarrow \langle 9\rangle$, so this rule can never be applied in a correct inference.

The following is a valid proof derivation, where $x, y \in \mathit{VarId}$ and $\sigma \in \mathit{State}$ with $\sigma(x) = \sigma(y) = 1$:

$$\dfrac{\dfrac{\cdot}{\langle x, \sigma\rangle \Downarrow \langle 1\rangle} \quad , \quad \dfrac{\dfrac{\cdot}{\langle y, \sigma\rangle \Downarrow \langle 1\rangle} \quad , \quad \dfrac{\cdot}{\langle x, \sigma\rangle \Downarrow \langle 1\rangle}}{\langle y * x, \sigma\rangle \Downarrow \langle 1\rangle} \quad , \quad \dfrac{\cdot}{\langle 2, \sigma\rangle \Downarrow \langle 2\rangle}}{\dfrac{\langle y * x + 2, \sigma\rangle \Downarrow \langle 3\rangle}{\langle x - (y * x + 2), \sigma\rangle \Downarrow \langle -2\rangle}}$$

The proof above can be regarded as an upside-down tree, with dots as leaves and instances of rule schemas as nodes. We call such "complete" (in the sense that their leaves are all dots and their nodes are correct rule instances) trees *proof trees*. This way, we have a way to mathematically *derive facts*, or *sequents*, about programs directly within their semantics. We may call the root of a proof tree the *fact (or sequent) that was proved or derived*, and the tree *its proof or derivation*.

Recall that our original intention was, for demonstration purposes, to attach various evaluation strategies to the arithmetic operations. We wanted `+` and `/` to be non-deterministic and `<=` to be left-right sequential; a non-deterministic evaluation strategy means that the sub-expressions are evaluated in any order, possibly interleaving their evaluation steps, which is different from non-deterministically picking an order and then evaluating the sub-expressions sequentially in that order. As an analogy, the former corresponds to evaluating the sub-expressions concurrently on a multithreaded machine, while the latter to non-deterministically queuing the sub-expressions and then evaluating them one by one on a sequential machine. The former has obviously potentially many more possible behaviors than the latter. Note that many programming languages opt for non-deterministic evaluation strategies for their expression constructs precisely to allow compilers to evaluate them concurrently; some language manuals explicitly warn the reader not to rely on any evaluation strategy of arithmetic constructs when writing programs.

Unfortunately, big-step semantics is not appropriate for defining non-deterministic evaluation strategies, because such strategies are, by their nature, small-step. One way to do it is to work with sets of values instead of with values and thus associate to each fragment of program in a state the set of all the values that it can non-deterministically evaluate to. However, such an approach would significantly complicate the big-step definition, so we prefer not to do it. Moreover, since IMP has no side effects (until Section 3.9), the non-deterministic evaluation strategies would not lead to non-deterministic results anyway (yet).

We next discuss the big-step rules for boolean expressions. The rule (BigStep-Bool) is similar to rule (BigStep-Int), but it has only two instances, one for t = `true` and one for t = `false`. The rules (BigStep-Not-True) and (BigStep-Not-False) are clear; they could have been combined into only one rule if we had assumed our "builtin" *Bool* equipped with a negation operation. Unlike the division, the conjunction has a short-circuited semantics: if the first conjunct evaluates to `false` then the entire conjunction evaluates to `false` (rule (BigStep-And-False)), and if the first conjunct evaluates to `true` then the conjunction evaluates to whatever truth value the second conjunct evaluates (rule (BigStep-And-True)).

Exercise 33. *Change the big-step semantics of the* IMP *conjunction so that it is not short-circuited.*

The role of statements in a language is to change the program state. Consequently, the rules for statements derive triples of the form $\langle s, \sigma\rangle \Downarrow \langle \sigma' \rangle$ with the meaning that if statement s is executed in state σ and *terminates*, then the resulting state is σ'. We will shortly discuss the aspect of termination in more detail. Rule (BigStep-Skip) states that `skip` does nothing with the state. (BigStep-Asgn) shows how the state σ gets updated by an assignment statement x `:=` a after a is evaluated in state σ using the rules for arithmetic expressions discussed above. (BigStep-Seq) shows how the state updates are propagated by the sequential composition of statements, and rules (BigStep-If-True) and (BigStep-If-False) show how the conditional first evaluates its condition and then, depending upon the truth value of that, it either evaluates its "then" branch or its "else" branch, but never both. The rules giving the big-step semantics of the while loop say that if the condition evaluates to `false` then the while loop dissolves and the state stays unchanged, and if the condition evaluates to `true` then the body followed by the very same while loop is evaluated (rule (BigStep-While-True)). Finally, (BigStep-Vars) gives the semantics of programs as the semantics of their statement in a state instantiating all the declared variables to 0. It all seems very natural, but there are some subtle aspects with regards to termination, which are discussed below.

On Proof Derivations, Evaluation and Termination

So far we have used the words "evaluation" and "termination" informally. In fact, without a formal definition of a programming language, there is no other way, but informal, to define these notions. Once one has a formal definition of a language, one can not only formally define important concepts like evaluation and termination, but can also rigorously reason about programs. We postpone the subject of program verification until Chapter 11; here we only define and discuss the other concepts.

Definition 16. *Given appropriate* IMP *configurations C and R, the* IMP *big-step sequent $C \Downarrow R$ is **derivable**, written* BigStep(IMP) $\vdash C \Downarrow R$, *iff there is some proof tree rooted in $C \Downarrow R$ which is derivable using the proof system* BigStep(IMP) *in Figure 3.8. Arithmetic (resp. boolean) expression $a \in AExp$ (resp. $b \in BExp$) **evaluates** to integer $i \in Int$ (resp. to truth value $t \in \{\texttt{true}, \texttt{false}\}$) in state $\sigma \in State$ iff* BigStep(IMP) $\vdash \langle a, \sigma\rangle \Downarrow \langle i\rangle$ *(resp.* BigStep(IMP) $\vdash \langle b, \sigma\rangle \Downarrow \langle t\rangle$*). Statement s **terminates** in state σ iff* BigStep(IMP) $\vdash \langle s, \sigma\rangle \Downarrow \langle \sigma'\rangle$ *for some $\sigma' \in State$; if that is the case, then we say that s **evaluates** in state σ to state σ', or that it **takes** state σ to state σ'. Finally, program p **terminates** iff* BigStep(IMP) $\vdash \langle p\rangle \Downarrow \langle \sigma\rangle$ *for some $\sigma \in State$.*

There are two reasons for which an IMP statement s may not terminate in a state σ: because it may contain a loop that does not terminate, or because it performs a division by zero and thus the rule (BigStep-Div) cannot apply. In the former case, the process of proof search does not terminate, while in the second case the process of proof search terminates in principle, but with a failure to find a proof. Unfortunately, big-step semantics cannot make any distinction between the two reasons for which a proof derivation cannot be found. If we want to "catch" division-by-zero within the semantics, then we need to add a special "error" value and propagate it through all the language constructs.

Exercise 34. *Add an* error *state and modify the big-step semantics in Figure 3.8 to allow derivations of sequents of the form $\langle s, \sigma\rangle \Downarrow \langle \mathsf{error}\rangle$ or $\langle p\rangle \Downarrow \langle \mathsf{error}\rangle$ when s evaluated in state σ or when p evaluated in the initial state performs a division by zero.*

A formal definition of a language allows to also formally define what it means for the language to be deterministic and to also prove it, as we do in the next result left as exercise:

Exercise 35. *Prove that the transition relation defined by the* BigStep(IMP) *proof system in Figure 3.8 is **deterministic**, that is:*

- *If* BigStep(IMP) $\vdash \langle a, \sigma\rangle \Downarrow \langle i\rangle$ *and* BigStep(IMP) $\vdash \langle a, \sigma\rangle \Downarrow \langle i'\rangle$ *then $i = i'$;*
- *If* BigStep(IMP) $\vdash \langle b, \sigma\rangle \Downarrow \langle t\rangle$ *and* BigStep(IMP) $\vdash \langle b, \sigma\rangle \Downarrow \langle t'\rangle$ *then $t = t'$;*
- *If s terminates in σ then there is a unique σ' such that* BigStep(IMP) $\vdash \langle s, \sigma\rangle \Downarrow \langle \sigma'\rangle$*;*
- *If p terminates then there is a unique σ such that* BigStep(IMP) $\vdash \langle s\rangle \Downarrow \langle \sigma\rangle$.

Prove the same results above for the proof system detecting division-by-zero as in Exercise 34.

Since each rule schema comprises a *recursively enumerable* collection of concrete instance rules, and since our language definition consists of a finite set of rule schemas, by enumerating all the concrete instances of these rules we get a recursively enumerable set of concrete instance rules. Furthermore, since proof trees built with nodes in a recursively enumerable set are themselves

recursively enumerable, it follows that the set of proof trees derivable with the proof system in Figure 3.8 is recursively enumerable. In other words, we can find an algorithm that enumerates all the proof trees, in particular one that enumerates all the derivable sequents $C \Downarrow R$. By enumerating all proof trees, given an IMP program p that terminates, one can eventually find the unique state σ such that $\langle p \rangle \Downarrow \langle \sigma \rangle$ is derivable. This simple-minded algorithm may take a very long time and a huge amount of resources, but it is theoretically important to understand that it can be done.

Exercise 36. *Show that there is no algorithm, based on proof derivation like above or on anything else, which takes as input an* IMP *program and says whether it terminates* ***or not****.*

The above follows from the fact that our simple language, due to its while loops and arbitrarily large integers, is Turing-complete. Thus, if one were able to decide termination of programs in our language then one were able to also decide termination of Turing machines, contradicting one of the basic undecidable problems, the *halting problem* (see Section 4.1 for more on Turing machines).

An interesting observation here is that non-termination of a program corresponds to *lack of proof*, and that the latter is not decidable in many interesting logics. Indeed, in *complete* logics, that is logics that admit a complete proof system, one can enumerate all the truths. However, in general there is not much one can do about non-truths, because the enumeration algorithm will loop forever when run on a non-truth. In decidable logics one can enumerate both truths and non-truths; clearly, decidable logics are not powerful enough for our task of defining programming languages, exactly because of the halting problem argument above.

3.2.3 Big-Step Operational Semantics in Rewriting Logic

Due to its straightforward recursive nature, big-step semantics is typically easy to represent in other formalisms and also easy to translate into interpreters for the defined languages in any programming language. (The difficulty with big-step semantics is to actually give big-step semantics to complex constructs, as discussed in Section 3.9.) It should therefore come at no surprise to the reader that one can associate a conditional rewrite rule to each big-step rule and hereby obtain a rewrite logic theory that faithfully captures the big-step definition.

In this section we first show that any big-step operational semantics BigStep can be mechanically translated into a rewriting logic theory $\mathcal{R}_{\text{BigStep}}$ in such a way that the corresponding derivation relations are step-for-step equivalent, that is, $\text{BigStep} \vdash C \Downarrow R$ if and only if $\mathcal{R}_{\text{BigStep}} \vdash \mathcal{R}_{C \Downarrow R}$, where $\mathcal{R}_{C \Downarrow R}$ is the corresponding syntactic translation of the big-step sequent $C \Downarrow R$ into a (one-step) rewrite rule. Second, we apply our generic translation technique on the big-step operational semantics BigStep(IMP) and obtain a rewriting logic semantics of IMP that is step-for-step equivalent to the original big-step semantics of IMP. Finally, we show how $\mathcal{R}_{\text{BigStep(Imp)}}$ can be seamlessly defined in Maude, thus yielding an interpreter for IMP essentially for free.

Faithful Embedding of Big-Step Operational Semantics into Rewriting Logic

To define our translation generically, we need to make some assumptions about the existence of an algebraic axiomatization of configurations. More precisely, we assume that for any parametric configuration C, $\overline{C}$ is an equivalent algebraic variant of C. By "parametric" configuration we mean a configuration that may possibly make use of parameters, such as $a \in \mathit{AExp}$, $\sigma \in \mathit{State}$, etc. By "equivalent" algebraic variant we mean a term of sort *Configuration* over an appropriate signature of

configurations like the one that we defined for IMP in Section 3.2.1 (see Figure 3.6); moreover, each "parameter" in C gets replaced by a *variable* of corresponding sort in $\overline{C}$.

Consider now a general-purpose big-step rule of the form

$$\frac{C_1 \Downarrow R_1,\ C_2 \Downarrow R_2,\ \ldots,\ C_n \Downarrow R_n}{C \Downarrow R}$$

where C, C_1, C_2, ..., C_n are configurations holding fragments of program together with all the needed semantic components, and R, R_1, R_2, ..., R_n are result configurations. As one may probably expect by now, we translate it into the following rewrite logic rule:

$$\overline{C} \to \overline{R} \ \textbf{ if } \ \overline{C_1} \to \overline{R_1} \ \wedge \ \overline{C_2} \to \overline{R_2} \ \wedge \ \cdots \ \wedge \ \overline{C_n} \to \overline{R_n}.$$

We make the reasonable assumption that configurations in BigStep are not nested.

Theorem 2. ***(Faithful embedding of big-step operational semantics into rewrite logic)*** *For any big-step operational semantics definition* BigStep*, and any* BigStep *appropriate configuration C and result configuration R, the following equivalence holds*

$$\text{BigStep} \ \vdash C \Downarrow R \iff \mathcal{R}_{\text{BigStep}} \vdash \mathcal{R}_{C \Downarrow R},$$

where $\mathcal{R}_{\text{BigStep}}$ is the rewrite logic semantic definition obtained from BigStep *translating each rule in* BigStep *as above, and where $\mathcal{R}_{C \Downarrow R}$ is the rewrite relation $\overline{C} \to^1 \overline{R}$. (Recall from Section 2.7 that $\to^1$ is the one-step rewriting relation obtained by dropping the transitivity rule of rewrite logic. Also, as C and R are parameter-free —parameters only appear in rules— $\overline{C}$ and $\overline{R}$ are ground terms.)*

The non-nestedness assumption on configurations in BigStep guarantees that the resulting rewrite rules in $\mathcal{R}_{\text{BigStep}}$ only apply at the top of the term they rewrite. Since one typically perceives "parameters" as "variables" anyway, the only apparent difference between BigStep and $\mathcal{R}_{\text{BigStep}}$ is the different notational conventions they use ("$\to$" instead of "$\Downarrow$" and conditional rewrite rules instead of conditional deduction rules). As Theorem 2 shows, there is a one-to-one correspondence also between their corresponding "computations" (or executions, or derivations). Therefore, $\mathcal{R}_{\text{BigStep}}$ *is* the big-step operational semantics BigStep, and *not* an "encoding" of it.

At our knowledge, there is no rewrite engine that supports the one-step rewrite relation $\to^1$ (that appears in Theorem 2). Indeed, rewrite engines aim at high-performance implementations of the general rewrite relation $\to$, which may even involve parallel rewriting (see Section 2.7 for the precise definitions of $\to^1$ and $\to$); $\to^1$ is meaningful only from a theoretical perspective and there is little to no practical motivation for an efficient implementation of it. Therefore, in order to execute the rewrite theory $\mathcal{R}_{\text{BigStep}}$ resulting from the mechanical translation of big-step semantics BigStep, one needs to take some precautions to ensure that $\to^1$ is actually identical to $\to$.

A sufficient condition ensuring that $\to^1$ is the same as $\to$ is that the configurations C appearing to the left of "$\Downarrow$" are always distinct from those to the right of "$\Downarrow$". More generally, if one makes sure that result configurations never appear as left hand sides of rules in $\mathcal{R}_{\text{BigStep}}$, then one is guaranteed that it is never the case that more than one rewrite step will ever be applied on a given configuration.

Corollary 1. *Under the same hypotheses as in Theorem 2, if result configurations never appear as left hand sides of rules in $\mathcal{R}_{\text{BigStep}}$, then*

$$\text{BigStep} \ \vdash C \Downarrow R \iff \mathcal{R}_{\text{BigStep}} \vdash \overline{C} \to^1 \overline{R} \iff \mathcal{R}_{\text{BigStep}} \vdash \overline{C} \to \overline{R}.$$

Fortunately, in our concrete big-step semantics of IMP, BigStep(IMP), the configurations to the left of "$\Downarrow$" and the result configurations to the right of "$\Downarrow$" are always distinct. Unfortunately, in general that may not always be the case. For example, when we extend IMP with side effects in Section 3.9, the (possibly affected) state also needs to be part of result configurations, so the semantics of integers is going to be given by an unconditional rule of the form $\langle i, \sigma \rangle \Downarrow \langle i, \sigma \rangle$, which after translation becomes the rewrite rule $\langle i, \sigma \rangle \to \langle i, \sigma \rangle$. This rule will make the rewrite relation $\to$ not terminate anymore (although the relation $\to^1$ terminates). There are at least two simple ways to ensure the hypothesis of Corollary 1:

1. It is highly expected that the only big-step rules in BigStep having a result configuration to the left of $\Downarrow$ are unconditional rules of the form $R \Downarrow R$; such rules typically say that a value is already evaluated. If that is the case, then one can simply drop all the corresponding rules $\overline{R} \to \overline{R}$ from $\mathcal{R}_{\text{BigStep}}$ and the resulting rewrite theory, say $\mathcal{R}'_{\text{BigStep}}$ still has the property $\text{BigStep} \vdash C \Downarrow R \iff \mathcal{R}'_{\text{BigStep}} \vdash \overline{C} \to \overline{R}$, which is desirable in order to execute the big-step definition on rewrite engines, although the property $\text{BigStep} \vdash C \Downarrow R \iff \mathcal{R}'_{\text{BigStep}} \vdash \overline{C} \to^1 \overline{R}$ will not hold anymore, because, e.g., even though $R \Downarrow R$ is a rule in BigStep, it is not the case that $\mathcal{R}'_{\text{BigStep}} \vdash \overline{R} \to^1 \overline{R}$.

2. If BigStep contains pairs $R' \Downarrow R$ where R' and R are possibly different result configurations, then one can apply the following general procedure. Change or augment the syntax of the configurations to the left or to the right of "$\Downarrow$", so that those changed or augmented configurations will always be different from the other ones. This is the technique employed in our representation of small-step operational semantics in rewriting logic in Section 3.3. More precisely, we prepend all the configurations to the left of the rewrite relation in $\mathcal{R}_{\text{BigStep}}$ with a circle "$\circ$", e.g., $\circ C \to R$, with the intuition that the circled configurations are "active", while the other ones are "inactive".

Regardless of how the desired property "$\text{BigStep} \vdash C \Downarrow R \iff \mathcal{R}_{\text{BigStep}} \vdash \overline{C} \to \overline{R}$" is ensured, note that, unfortunately, $\mathcal{R}_{\text{BigStep}}$ lacks the main strengths of rewriting logic that make it a good formalism for concurrency: in rewriting logic, rewrite rules can apply under any context and in parallel. Indeed, the rules of $\mathcal{R}_{\text{BigStep}}$ can only apply at the top, sequentially.

Big-Step Operational Semantics of IMP in Rewriting Logic

Figure 3.9 gives the rewriting logic theory $\mathcal{R}_{\text{BigStep(IMP)}}$ that is obtained by applying the procedure above to the big-step semantics of IMP, BigStep(IMP), in Figure 3.8. We have used the rewriting logic convention that variables start with upper-case letters. For the state variable, we used σ, that is, a larger σ symbol. Besides the parameter vs. variable subtle (but not unexpected) aspect, the only perceivable difference between BigStep(IMP) and $\mathcal{R}_{\text{BigStep(IMP)}}$ is the different notational conventions they use ("$\to$" instead of "$\Downarrow$" and conditional rewrite rules instead of conditional deduction rules). The following corollary of Theorem 2 and Corollary 1 establishes the faithfulness of the representation of the big-step operational semantics of IMP in rewriting logic:

Corollary 2. $\text{BigStep(IMP)} \vdash C \Downarrow R \iff \mathcal{R}_{\text{BigStep(IMP)}} \vdash \overline{C} \to \overline{R}$.

Therefore, there is no perceivable computational difference between the IMP-specific proof system BigStep(IMP) and generic rewriting logic deduction using the IMP-specific rewrite rules in $\mathcal{R}_{\text{BigStep(IMP)}}$, so the two are faithfully equivalent.

$$
\begin{array}{rcl}
\langle I, \sigma\rangle \to \langle I\rangle & & \\
\langle X, \sigma\rangle \to \langle \sigma(X)\rangle & & \\
\langle A_1 \texttt{ + } A_2, \sigma\rangle \to \langle I_1 \ +_{Int}\ I_2\rangle & \textbf{if} & \langle A_1, \sigma\rangle \to \langle I_1\rangle \ \wedge\ \langle A_2, \sigma\rangle \to \langle I_2\rangle \\
\langle A_1 \texttt{ / } A_2, \sigma\rangle \to \langle I_1 \ /_{Int}\ I_2\rangle & \textbf{if} & \langle A_1, \sigma\rangle \to \langle I_1\rangle \ \wedge\ \langle A_2, \sigma\rangle \to \langle I_2\rangle \ \wedge\ I_2 \neq 0 \\
\langle A_1 \texttt{ <= } A_2, \sigma\rangle \to \langle I_1 \ \leq_{Int}\ I_2\rangle & \textbf{if} & \langle A_1, \sigma\rangle \to \langle I_1\rangle \ \wedge\ \langle A_2, \sigma\rangle \to \langle I_2\rangle \\
\langle T, \sigma\rangle \to \langle T\rangle & & \\
\langle \texttt{not } B, \sigma\rangle \to \langle \texttt{false}\rangle & \textbf{if} & \langle B, \sigma\rangle \to \langle \texttt{true}\rangle \\
\langle \texttt{not } B, \sigma\rangle \to \langle \texttt{true}\rangle & \textbf{if} & \langle B, \sigma\rangle \to \langle \texttt{false}\rangle \\
\langle B_1 \texttt{ and } B_2, \sigma\rangle \to \langle \texttt{false}\rangle & \textbf{if} & \langle B_1, \sigma\rangle \to \langle \texttt{false}\rangle \\
\langle B_1 \texttt{ and } B_2, \sigma\rangle \to \langle T\rangle & \textbf{if} & \langle B_1, \sigma\rangle \to \langle \texttt{true}\rangle \ \wedge\ \langle B_2, \sigma\rangle \to \langle T\rangle \\
\langle \texttt{skip}, \sigma\rangle \to \langle \sigma\rangle & & \\
\langle X \texttt{ := } A, \sigma\rangle \to \langle \sigma[I/X]\rangle & \textbf{if} & \langle A, \sigma\rangle \to \langle I\rangle \\
\langle S_1 \texttt{;} S_2, \sigma\rangle \to \langle \sigma_2\rangle & \textbf{if} & \langle S_1, \sigma\rangle \to \langle \sigma_1\rangle \ \wedge\ \langle S_2, \sigma_1\rangle \to \langle \sigma_2\rangle \\
\langle \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2, \sigma\rangle \to \langle \sigma_1\rangle & \textbf{if} & \langle B, \sigma\rangle \to \langle \texttt{true}\rangle \ \wedge\ \langle S_1, \sigma\rangle \to \langle \sigma_1\rangle \\
\langle \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2, \sigma\rangle \to \langle \sigma_2\rangle & \textbf{if} & \langle B, \sigma\rangle \to \langle \texttt{false}\rangle \ \wedge\ \langle S_2, \sigma\rangle \to \langle \sigma_2\rangle \\
\langle \texttt{while } B \texttt{ do } S, \sigma\rangle \to \langle \sigma\rangle & \textbf{if} & \langle B, \sigma\rangle \to \langle \texttt{false}\rangle \\
\langle \texttt{while } B \texttt{ do } S, \sigma\rangle \to \langle \sigma'\rangle & \textbf{if} & \langle B, \sigma\rangle \to \langle \texttt{true}\rangle \ \wedge\ \langle S \texttt{;while } B \texttt{ do } S, \sigma\rangle \to \langle \sigma'\rangle \\
\langle \texttt{ vars } Xl \texttt{ ; } S\rangle \to \langle \sigma\rangle & \textbf{if} & \langle S, Xl \mapsto 0\rangle \to \langle \sigma\rangle
\end{array}
$$

Figure 3.9: $\mathcal{R}_{\text{BIGSTEP(IMP)}}$: the Big-Step Operational Semantics of IMP in Rewriting Logic

☆ Maude Definition of IMP Big-Step Operational Semantics

Figure 3.10 shows a straightforward Maude representation of the rewrite theory $\mathcal{R}_{\text{BIGSTEP(IMP)}}$ in Figure 3.9. The Maude module `IMP-SEMANTICS-BIGSTEP` in Figure 3.10 is executable, so Maude, through its rewriting capabilities, yields an interpreter for IMP; for example, the command

```
Maude> rewrite < sumPgm > .
```

where `sumPgm` is the first program defined in the module `IMP-PROGRAMS` in Figure 3.4, produces a result of the form (the exact statistics are irrelevant here, so they were replaced by "..."):

```
rewrites: 4019 in ... cpu (... real) (... rewrites/second)
result Configuration: < n |-> 0 , s |-> 5050 >
```

The obtained interpreter actually has acceptable performance; for example, all the programs in Figure 3.4 together take a fraction of a second to execute on conventional PCs or laptops.

Once one has a rewriting logic definition in Maude, one can use any of the general-purpose tools provided by Maude on that definition; the rewrite engine is only one of them. For example, one can exhaustively search for all possible behaviors of a program using the `search` command:

```
Maude> search < sumPgm > =>! Cfg:Configuration .
```

Since our IMP language so far is deterministic, the `search` command will not discover any new behaviors. However, as shown in Section 3.8 where we extend IMP with various language features, the `search` command can indeed show all the behaviors of a non-deterministic program.

```
mod IMP-SEMANTICS-BIGSTEP is including IMP-CONFIGURATIONS-BIGSTEP .
  var X : VarId . var I I1 I2 : Int . var A A1 A2 : AExp . var B B1 B2 : BExp . var T : Bool .
  var S S1 S2 : Stmt . var Xl : List{VarId} . var Sigma Sigma' Sigma1 Sigma2 : State .

  rl < I,Sigma > => < I > .

  rl < X,Sigma > => < Sigma(X) > .

 crl < A1 + A2,Sigma > => < I1 +Int I2 >
  if < A1,Sigma > => < I1 > /\ < A2,Sigma > => < I2 > .

 crl < A1 / A2,Sigma > => < I1 /Int I2 >
  if < A1,Sigma > => < I1 > /\ < A2,Sigma > => < I2 > /\ I2 =/= 0 .

  rl < T,Sigma > => < T > .

 crl < A1 <= A2,Sigma > => < I1 <=Int I2 >
  if < A1,Sigma > => < I1 > /\ < A2,Sigma > => < I2 > .

 crl < not B,Sigma > => < false > if < B,Sigma > => <  true > .

 crl < not B,Sigma > => <  true > if < B,Sigma > => < false > .

 crl < B1 and B2,Sigma > => < false > if < B1,Sigma > => < false > .

 crl < B1 and B2,Sigma > => < T > if < B1,Sigma > => < true > /\ < B2,Sigma > => < T > .

  rl < skip,Sigma > => < Sigma > .

 crl < X := A,Sigma > => < Sigma[I / X] > if < A,Sigma > => < I > .

 crl < S1 ; S2,Sigma > => < Sigma2 >
  if < S1,Sigma > => < Sigma1 > /\ < S2,Sigma1 > => < Sigma2 > .

 crl < if B then S1 else S2,Sigma > => < Sigma1 >
  if < B,Sigma > => < true  > /\ < S1,Sigma > => < Sigma1 > .

 crl < if B then S1 else S2,Sigma > => < Sigma2 >
  if < B,Sigma > => < false > /\ < S2,Sigma > => < Sigma2 > .

 crl < while B do S,Sigma > => < Sigma > if < B,Sigma > => < false > .

 crl < while B do S,Sigma > => < Sigma' >
  if < B,Sigma > => < true > /\ < S ; while B do S,Sigma > => < Sigma' > .

 crl < vars Xl ; S > => < Sigma > if < S, (Xl |-> 0) > => < Sigma > .
endm
```

Figure 3.10: The Big-Step Operational Semantics of IMP in Maude

3.2.4 Notes

Big-step operational semantics was introduced under the name *natural semantics* by Kahn [17] in 1987, in the context of defining Mini-ML, a simple pure version of the ML language. Natural semantics is indeed very natural when defining pure, sequential and structured languages, so it quickly became very popular. However, as Kahn himself noted in his seminal paper, the idea of using proof systems to capture the operational semantics of programming languages goes back to Plotkin [36] in 1981, under the name (small-step) *structural operational semantics* (SOS); we will discuss SOS in depth in Section 3.3. Kahn and others found big-step semantics more natural and convenient than Plotkin's SOS, essentially because it is more abstract and denotational in nature, and one needs fewer rules to define a language semantics.

One of the most notable uses of natural semantics is the formal semantics of Standard ML by Milner *et al.* [30]. After twenty years of natural semantics, it is now common wisdom that big-step semantics is inappropriate as a rigorous formalism for defining languages with complex features such as exceptions or concurrency. To give a reasonably compact and readable definition of Standard ML in [30], Milner *et al.* had to make several informal notational conventions, such a "store convention" to avoid having to mention the store in every rule, and an "exception convention" to avoid having to double the number of rules for the sole purpose of supporting exceptions. As rightfully noticed by Mosses [33], such conventions are not only adhoc and language specific, but may also lead to erroneous definitions. Section 3.8 illustrates in detail the limitations of big-step operational semantics. The most common use of natural semantics these days is to define static semantics of programming languages and calculi, such as, for example, type systems.